4.7 Nested For Loops

When we introduced for loops, we said that the loop body consists of one of more statements. We saw in 4.5 For Loop Variations that we could put if statements inside loop bodies. In this section, we'll see that a for loop body can itself contain another for loop, since for loops are themselves statements. We'll study uses of these *nested for loops*, and also draw comparisons between them and comprehensions from the previous chapter.

Nested loops and nested data

Nested loops are particularly useful when dealing with nested data. As a first example, suppose we have a list of lists of integers:

```
>>> lists_of_numbers = [[1, 2, 3], [10, -5], [100]]
```

Our goal is to compute the sum of all of the elements of this list:

```
def sum_all(lists_of_numbers: list[list[int]]) -> int:
    """Return the sum of all the numbers in the given lists_of_numbers.

    >>> sum_all([[1, 2, 3], [10, -5], [100]])
    111
    """
```

We can start with our basic loop accumulator pattern:

```
def sum_all(lists_of_numbers: list[list[int]]) -> int:
    """..."""
    # ACCUMULATOR sum_so_far: keep track of the running sum of the numbers.
    sum_so_far = 0

    for ... in lists_of_numbers:
        sum_so_far = ...

    return sum_so_far
```

The difference between this function and in `my_sum` from 4.4 is that here our loop variable in `for ... in lists_of_numbers` does not refer to a single number, but rather a list of numbers:

```python
def sum_all(lists_of_numbers: list[list[int]]) -> int:
    """..."""
    # ACCUMULATOR sum_so_far: keep track of the running sum of the numbers.
    sum_so_far = 0

    for numbers in lists_of_numbers:  # numbers is a list of numbers, not a
     single number!
        sum_so_far = ...

    return sum_so_far
```

So here is one way of completing this function, by using the builtin `sum` function:

```python
def sum_all(lists_of_numbers: list[list[int]]) -> int:
    """..."""
    # ACCUMULATOR sum_so_far: keep track of the running sum of the numbers.
    sum_so_far = 0

    for numbers in lists_of_numbers:  # numbers is a list of numbers, not a
     single number!
        sum_so_far = sum_so_far + sum(numbers)

    return sum_so_far
```

This implementation is structurally similar to the `my_sum` implementation we had in Section 4.4. But how would we implement this function *without* using `sum`? For this we need another for loop:

```python
def sum_all(lists_of_numbers: list[list[int]]) -> int:
    """..."""
    # ACCUMULATOR sum_so_far: keep track of the running sum of the numbers.
    sum_so_far = 0

    for numbers in lists_of_numbers:  # numbers is a list of numbers, not a
     single number!
        for number in numbers:        # number is a single number
            sum_so_far = sum_so_far + number

    return sum_so_far
```

We say that the `for number in numbers` loops is *nested* within the `for numbers in lists_of_numbers`. What happens when we call our doctest example, `sum_all([[1, 2, 3], [10, -5], [100]])`? Let's break this down step by step.

1. First, the assignment statement `sum_so_far = 0` executes, creating our accumulator variable.

2. The outer loop is reached.

    - The loop variable `list_of_numbers` is assigned the first element in `lists_of_numbers`, which is `[1, 2, 3]`.

    - Then, the body of the outer loop is executed. Its body is just one statement: the inner for loop, `for number in numbers`.

        - The inner loop variable `number` is assigned the first value in `numbers`, which is `1`.

        - The inner loop body gets executed, updating the accumulator. `sum_so_far` is reassigned to `1` (since `0 + 1 == 1`).

        - The inner loop iterates twice more, for `number = 2` and `number = 3`.[1] At

          [1] Notice that `numbers` is the *same value (`[1, 2, 3]`) for this entire part.

          each iteration, the accumulator is updated, first by adding `2` and then `3`. At this point, `sum_so_far = 6` (`0 + 1 + 2 + 3`).

        - After all three iterations of the inner loop occur, the inner loop stops. The Python interpreter is done executing this statement.

    - The next iteration of the *outer loop* occurs; `numbers` is assigned to the list `[10, -5]`.

    - Again, the body of the outer loop occurs.

        - The inner loop now iterates twice: for `number = 10` and `number = -5`. `sum_so_far` is reassigned twice more, with a final value of `11` (`6 + 10 + -5`).

    - The outer loop iterates one more time, for `numbers = [100]`.

    - Again, the body of the outer loop occurs.

        - The inner loop iterates once, for `number = 100`. `sum_so_far` is reassigned to `111` (`11 + 100`).

    - At last, there are no more iterations of the outer loop, and so it stops.

3. After the outer loop is done, the `return` statement executes, returning the value of `sum_so_far`, which is `111`.

Whew, that's a lot of writing! We can summarize the above behaviour by creating a *loop accumulation table*. Note that the table below has the same structure as the ones we've seen before, but is more complex because its columns include both the outer and inner loop variables and iterations. The `accumulator` column shows the value of `sum_so_far` at the *end* of the iteration of the inner loop. Pay close attention to the *order* of the rows, as this matches the order of execution we described above.

Outer loop iteration	Outer loop variable (list_of_numbers)	Inner loop iteration	Inner loop variable (number)	Accumulator (sum_so_far)
0				0
1	[1, 2, 3]	0		0
1	[1, 2, 3]	1	1	1
1	[1, 2, 3]	2	2	3
1	[1, 2, 3]	3	3	6
2	[10, -5]	0		6
2	[10, -5]	1	10	16
2	[10, -5]	2	-5	11
3	[100]	0		11
3	[100]	1	100	111

The Cartesian product

Our next example illustrates how to use nested loops on two different collections, obtaining all pairs of possible values from each collection. If that sounds familiar, well, it should be!

```
def product(set1: set, set2: set) -> set[tuple]:
    """Return the Cartesian product of set1 and set2.

    >>> result = product({10, 11}, {5, 6, 7})
    >>> result == {(10, 5), (10, 6), (10, 7), (11, 5), (11, 6), (11, 7)}
    True
    """
```

Before we get to writing any loops at all, let's remind ourselves how we would write a comprehension to compute the Cartesian product:

```
>>> set1 = {10, 11}
>>> set2 = {5, 6, 7}
>>> result = {(x, y) for x in set1 for y in set2}
>>> result == {(10, 5), (10, 6), (10, 7), (11, 5), (11, 6), (11, 7)}
True
```

Now we'll see how to write this using nested for loop:

```
def cartesian_product(set1: set, set2: set) -> set[tuple]:
    """Return the Cartesian product of set1 and set2.

    >>> result = cartesian_product({10, 11}, {5, 6, 7})
    >>> result == {(10, 5), (10, 6), (10, 7), (11, 5), (11, 6), (11, 7)}
    True
    """
    # ACCUMULATOR product_so_far: keep track of the tuples from the pairs
    # of elements visited so far.
    product_so_far = set()

    for x in set1:
        for y in set2:
            product_so_far = set.union(product_so_far, {(x, y)})

    return product_so_far
```

As we saw in our first example, here the inner loop `for y in set2` iterates through every element of `set2` for every element of `x` in `set1`. You can visualize this in the following loop accumulation table:

Outer loop iteration	Outer loop var (x)	Inner loop iteration	Inner loop var (y)	Accumulator (`product_so_far`)
0				`set()`
1	`10`	0		`set()`
1	`10`	1	`5`	`{(10, 5)}`
1	`10`	2	`6`	`{(10, 5), (10, 6)}`
1	`10`	3	`7`	`{(10, 5), (10, 6), (10, 7)}`
2	`11`	0		`{(10, 5), (10, 6), (10, 7)}`
2	`11`	1	`5`	`{(10, 5), (10, 6), (10, 7), (11, 5)}`
2	`11`	2	`6`	`{(10, 5), (10, 6), (10, 7), (11, 5), (11, 6)}`
2	`11`	3	`7`	`{(10, 5), (10, 6), (10, 7), (11, 5), (11, 6), (11, 7)}`

Another way of visualizing the return value is:

```
{
    (10, 5), (10, 6), (10, 7),  # First three tuples are from the first
         iteration of the outer loop
    (11, 5), (11, 6), (11, 7)   # Next three tuples are from the second
         iteration of the outer loop
}
```

Outer and inner accumulators

Both the `sum_all` and `cartesian_product` examples we've seen so far have used a single accumulator that is updated inside the inner loop body. However, *each loop* can have its own accumulator (and in fact, more than one accumulator). This is more complex, but offers more flexibilty than a single accumulator does alone.

As an example, suppose we have a list of lists of integers called `grades`. Each element of `grades` corresponds to a course and contains a list of grades obtained in that course. Let's see an example of the data:

```
>>> grades = [
...     [70, 75, 80],       # ENG196
...     [70, 80, 90, 100],  # CSC110
...     [80, 100]           # MAT137
... ]
```

Notice how the list of grades for course ENG196 does not have the same length as CSC110 or MAT137. Our goal is to return a new list containing the *average grade* of each course. We saw in Section 4.5 how to use loops to calculate the average of a collection of numbers:

```
def average(numbers: Iterable[int]) -> float:
    """Return the average of a collection of integers.

    Preconditions:
        - len(numbers) > 0
    """
    # ACCUMULATOR len_so_far: keep track of the number of elements seen so
         far in the loop.
    len_so_far = 0
    # ACCUMULATOR total_so_far: keep track of the total of the elements seen
         so far in the loop.
    total_so_far = 0

    for number in numbers:
        len_so_far = len_so_far + 1
        total_so_far = total_so_far + number

    return total_so_far / len_so_far
```

We can calculate a list of averages for each course using a comprehension:[2]

[2] Exercise: write a precondition expression to guarantee there are no empty lists in `grades`.

```python
def course_averages_v1(grades: list[list[int]]) -> list[float]:
    """Return a new list for which each element is the average of the grades
    in the inner list at the corresponding position of grades.

    >>> course_averages_v1([[70, 75, 80], [70, 80, 90, 100], [80, 100]])
    [75.0, 85.0, 90.0]
    """
    return [average(course_grades) for course_grades in grades]
```

We can translate this into a for loop using a list accumulator variable and list concatenation for the update:

```python
def course_averages_v2(grades: list[list[int]]) -> list[float]:
    """Return a new list for which each element is the average of the grades
    in the inner list at the corresponding position of grades.

    >>> course_averages_v2([[70, 75, 80], [70, 80, 90, 100], [80, 100]])
    [75.0, 85.0, 90.0]
    """
    # ACCUMULATOR averages_so_far: keep track of the averages of the lists
    # visited so far in grades.
    averages_so_far = []

    for course_grades in grades:
        course_average = average(course_grades)
        averages_so_far = averages_so_far + [course_average]

    return averages_so_far
```

Now let's see how to calculate the `course_average` variable for each course by using an inner loop instead of the `average` function. We can do this by *expanding the definition of* `average` directly in the loop body, with just a few minor tweaks:

```python
def course_averages_v3(grades: list[list[int]]) -> list[float]:
    """Return a new list for which each element is the average of the grades
    in the inner list at the corresponding position of grades.

    >>> course_averages_v3([[70, 75, 80], [70, 80, 90, 100], [80, 100]])
    [75.0, 85.0, 90.0]
    """
    # ACCUMULATOR averages_so_far: keep track of the averages of the lists
    # visited so far in grades.
    averages_so_far = []
```

```
    for course_grades in grades:
        # ACCUMULATOR len_so_far: keep track of the number of elements seen
         so far in course_grades.
        len_so_far = 0
        # ACCUMULATOR total_so_far: keep track of the total of the elements
         seen so far in course_grades.
        total_so_far = 0

        for grade in course_grades:
            len_so_far = len_so_far + 1
            total_so_far = total_so_far + grade

        course_average = total_so_far / len_so_far

        averages_so_far = averages_so_far + [course_average]

    return averages_so_far
```

It may be surprising to you that we can do this! Just as how in the last chapter we saw that we can take a predicate and expand it into its definition, we can do the same thing for Python functions with multiple statements in their body. The only change we needed to make was the return statement of `average`. The original function had the statement `return total_so_far / len_so_far`. Because our loop assigned this return value to `course_average`, we changed the code to:

```
course_average = total_so_far / len_so_far
```

One important note about the structure of this nested loop is that the inner loop accumulators are assigned to *inside* the body of the outer loop*, rather than at the top of the function body. This is because the accumulators `len_so_far` and `total_so_far` are specific to `course_grades`, which changes at each iteration of the outer loop. The statements `len_so_far = 0` and `total_so_far = 0` act to “reset” these accumulators for each new `course_grades` list.

Let’s take a look at our final loop accumulation table in this section, which illustrates the execution of `course_averages_v3([[70, 75, 80], [70, 80, 90, 100], [80, 100]])` and how each loop variable and accumulator changes. Please take your time studying this table carefully—it isn’t designed to be a “quick read”, but to really deepen your understand of what’s going on!

Outer loop iteration	**Outer loop variable (`course_grades`)**	**Inner loop iteration**	**Inner loop variable (`grade`)**	**Inner accumulator (`len_so_far`)**	**Inner accumulator (`total_so_far`)**	**Outer accumulator (`averages_so_far`)**
0						`[]`
1	`[70, 75, 80]`	0		`0`	`0`	`[]`
1	`[70, 75, 80]`	1	`70`	`1`	`70`	`[]`

Outer loop iteration	Outer loop variable (course_grades)	Inner loop iteration	Inner loop variable (grade)	Inner accumulator (len_so_far)	Inner accumulator (total_so_far)	Outer accumulator (averages_so_far)
1	[70, 75, 80]	2	75	2	145	[]
1	[70, 75, 80]	3	80	3	225	[75.0]
2	[70, 80, 90, 100]	0		0	0	[75.0]
2	[70, 80, 90, 100]	1	70	1	70	[75.0]
2	[70, 80, 90, 100]	2	80	2	150	[75.0]
2	[70, 80, 90, 100]	3	90	3	240	[75.0]
2	[70, 80, 90, 100]	4	100	4	340	[75.0, 85.0]
3	[80, 100]	0		0	0	[75.0, 85.0]
3	[80, 100]	1	80	1	80	[75.0, 85.0]
3	[80, 100]	2	100	2	180	[75.0, 85.0, 90.0]

Summary: understanding and simplifying nested for loops

Nested for loops are a powerful tool in our understanding of the Python programming language, but they are by far the most complex and most error-prone that we've studied so far. Just as we saw with nested expressions and nested if statements, nested loops have the potential to greatly increase the size and complexity of our code. Contrast the implementation of `course_averages_v3` against `course_averages_v2` (or `course_averages_v1`), for example.

While nested loops are sometimes inevitable or convenient, we recommend following these guidelines to simplify your use of nested loops to help you better understand your code:

1. Use nested loops when you have a single accumulator that can be initialized just once before the nested loop (e.g., `sum_all` and `cartesian_product`).
2. If you have a nested loop where the inner loop can be replaced by a built-in aggregation function (e.g., `sum` or `len`), use the built-in function instead.
3. If you have a nested loop where the inner loop has a separate accumulator that is assigned inside the outer loop (e.g., `course_averages_v3`), move the accumulator and inner loop into a new function, and call that function from within the original outer loop.

References

- CSC108 videos: Nested loops (Part 1, Part 2)